

YAUHENI NIKANOVICH

---

# MOBX FOR DUMMIES

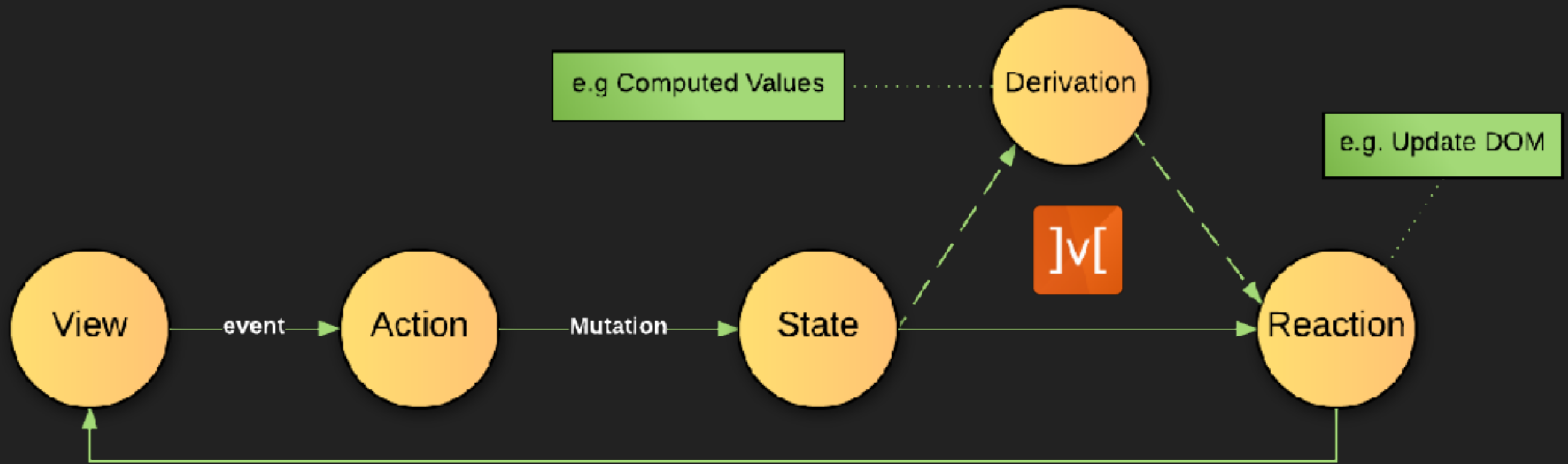


**MobX** is a simple, scalable and battle tested state management solution

The philosophy behind **MobX** is very simple: Anything that can be derived from the application state, should be derived. Automatically.

---

# FLOW



# REACT-MOBX

**React and MobX together are a powerful combination.**

- ▶ React renders the application state by providing mechanisms to translate it into a tree of renderable components.
- ▶ MobX provides the mechanism to store and update the application state that React then uses.

# Observable

**MobX** adds observable capabilities to existing data structures like objects, arrays and class instances.

Usage:

- ▶ `observable(value)`
- ▶ `@observable classProperty = value`

## ES NEXT

```
class PersonState {  
  @observable firstName  
  @observable lastName  
  
  constructor() {  
    this.firstName = ''  
    this.lastName = ''  
  }  
}  
  
export default new PersonState()
```

## ES5

```
var personState = mobx.observable({  
  firstName: '',  
  lastName: ''  
})
```

# Observer

**Observer** function / decorator can be used to turn ReactJS components into reactive components.

It wraps the component's render function in `mobx.autorun` to make sure that any data that is used during the rendering of a component forces a re-rendering upon change.

**Observer** is available through the separate `mobx-react` package.




# Observer

Usage:


- ▶ `observer(value)`
- ▶ `@observer classProperty = value`

## STATELESS



```
export const EventCard =
  inject('card', 'person')(observer(
    (props, { card, person }) =>
      <div className={classNames.CARD}>
        <h1>Preview:</h1>
        <div className={classNames.CARD_BOX}>
          <div className={classNames.CARD_BOX_HEADER}>
            <h2>{props.card.eventName}</h2>
          </div>
          <div className={classNames.CARD_BOX_BODY}>
            <h4>{props.person.fullName}</h4>
            <h4>{props.card.getTopic}</h4>
            <h4>{props.card.getDescription}</h4>
          </div>
          <div className={classNames.CARD_BOX_FOOTER}>
            <h2>{props.card.date}</h2>
          </div>
        </div>
      </div>
    ))
```

## STATEFULL



```
@inject('person') @observer
class App extends Component {
  render() {
    return (
      <div className={classNames.APP}>
        <EventForm person={this.props.person} />
        <EventCard />
        <DevTools />
      </div>
    )
  }
}
```

# Observer and store(s)

In order to bind a component and store(s) we can use `Provider` component.


`Provider` allows to inject passed stores from the child components by using props.

```
import PersonState from './store/PersonState'
import EventCardState from './store/EventCardState'

import App from './components/App'

const store = {
  person: PersonState,
  card: EventCardState,
}

render(
  <Provider { ...store }>
    <AppContainer>
      <App />
    </AppContainer>
  </Provider>,
  document.getElementById('root')
)
```



```
@inject('person') @observer
class App extends Component {
  render() {
    return (
      <div className={classNames.APP}>
        <EventForm person={this.props.person} />
        <EventCard />
        <DevTools />
      </div>
    )
  }
}
```



## Observer adds lifecycle hook

When using `mobx-react` you can define a new life cycle hook, `componentWillReact` that will be triggered when a component will be scheduled to re-render because data it observes has changed

- ▶ `componentWillReact` doesn't take arguments
- ▶ `componentWillReact` won't fire before the initial render (use `componentWillMount` instead)

# When to apply Observer

There is a simple rule: all components that render observable data

With `@observer` there is no need to distinguish 'smart' components from 'dumb' components for the purpose of rendering

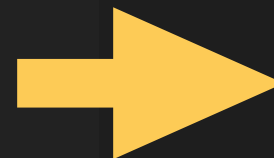
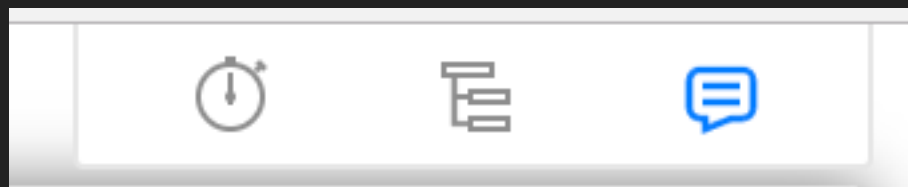
It makes sure that whenever you start using observable data, wrapped component will respond it



# MobX-React DevTools

In combination with `@observer` you can use the **MobX-React DevTools**, it shows exactly when your components are re-rendered, also it shows data dependencies of your components

```
npm i --save mobx-react-devtools
```



```
import DevTools from 'mobx-react-devtools'

@observer
class App extends Component {
  render() {
    return (
      <div className={classNames.APP}>
        <EventForm />
        <EventCard />
        <DevTools />
      </div>
    )
  }
}
```

# MobX-React DevTools



## VISUALIZE COMPONENT RE-RENDER

PersonState: 1x 0 / 11 ms

FirstName:

LastName:

EventCardState:

EventName:

Topic:

Date:

Description:

## SHOW COMPONENT DEPENDENCY TREE

```
<component>#0.render()
├── reactive props
├── EventCardState@3.eventName
├── PersonState@1.fullName
│   ├── PersonState@1.firstName
│   └── PersonState@1.lastName
├── EventCardState@3.getTopic
│   └── EventCardState@3.topic
├── EventCardState@3.getDescription
│   └── EventCardState@3.description
└── EventCardState@3.date
```

## LOG STATE CHANGES/REACTION

```
▶ action 'changeFirstName'
▶ reaction '<component>#0.render()'
▶ reaction '<component>#0.render()'
>
```

# Characteristics of Observer component

- ▶ **Observer** only subscribe to the data structures that were actively used during the last render. This means that you cannot under-subscribe or over-subscribe. You can even use data in your rendering that will only be available at later moment in time. This is ideal for asynchronously loading data.
- ▶ You are not required to declare what data a component will use. Instead, dependencies are determined at runtime and tracked in a very fine-grained manner.
- ▶ Usually reactive components have no or little state, as it is often more convenient to encapsulate (view) state in objects that are shared with other component. But you are still free to use native state.
- ▶ `@observer` implements `shouldComponentUpdate` in the same way as **PureRenderMixin** so that children are not re-rendered unnecessary.
- ▶ Reactive components sideways load data. Parent components won't re-render unnecessarily even when child components will.
- ▶ `@observer` does not depend on React's context system.

# Action

Any application has actions. Actions are anything that modify the state. With **MobX** you can make it explicit in your code where your actions live by marking them. Actions help you to structure your code better.

Action is advised to use on any function that modifies Observables or has side effects.

Action also provides useful debugging information in combination with **DevTools**



Action using action is mandatory when strict mode is enabled

# Action

Action takes a function as argument and returns it after wrapping it with `untracked`, `transaction` and `allowStateChanges`

Usage:

- ▶ `action(fn)`
- ▶ `action(name, fn)`
- ▶ `@action classMethod()`
- ▶ `@action(name) classMethod()`

```
const changeLastName = action(function(val) {  
  someStore.lastName = val  
})
```

```
@action changeLastName(val) {  
  this.lastName = val  
}
```

# Transaction

Transaction is a kind of scope of operations. Observers will not notified before transaction will not done

Note that transaction runs completely synchronously

Transactions can be nested. Only after completing the outer transaction pending reactions will be run

## allowStateChanges

Can be used to (dis)allow state changes in a certain function. Used internally by action to allow changes, and by computed and observer to disallow state changes.



# Strict mode

Usage:

- ▶ `useStrict(boolean)`

Enables / disables strict mode *globally*.

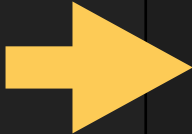
In strict mode, it is not allowed to change any state outside of an action

Violation consequence:

```
▶ Uncaught Error: [mobx] Invariant failed: Since mobx.js:2734  
strict-mode is enabled, changing observed observable values  
outside actions is not allowed. Please wrap the code in an  
`action` if this change is intended. Tried to modify:  
PersonState@1.firstName
```

# Computed

With **MobX** you can define values that will be derived automatically when relevant data is modified. By using the `@computed` decorator



```
import { observable, computed } from 'mobx'

class PersonState {
  @observable firstName
  @observable lastName

  constructor() {
    this.firstName = ''
    this.lastName = ''
  }

  @computed get fullName() {
    if ([this.firstName, this.lastName].some(item => item.length > 0))
      return `Speaker: ${this.firstName} ${this.lastName}`
  }
}

export default new PersonState()
```

# PropTypes

`mobx-react` provides the following additional `PropTypes` which can be used to validate against **MobX** structures:

- ▶ `observableArray`
- ▶ `observableArrayOf(React.PropTypes.number)`
- ▶ `observableMap`
- ▶ `observableObject`
- ▶ `arrayOrObservableArray`
- ▶ `arrayOrObservableArrayOf(React.PropTypes.number)`
- ▶ `objectOrObservableObject`

Import `PropTypes` from `mobx-react` and use it:

- ▶ `PropTypes.observableArray`

# PROS AND CONS

## Pros:

- ▶ Makes your components reactive
- ▶ Easy to understand
- ▶ Friendly for users with OOP background
- ▶ Less code
- ▶ Steep learning curve
- ▶ Responsive community
- ▶ ?? Magic ??

## Cons:

?? Magic ??

Q & A

# Thanks for attention

YAUHENI NIKANOVICH

---

SLIDES: [GOO.GL/2EUFCF](https://goo.gl/2EUFCF)

EMAIL: [ZHENYANIKON@GMAIL.COM](mailto:ZHENYANIKON@GMAIL.COM)